Job Scheduling in a Heterogeneous Grid Environment

Hongzhang Shan
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

Warren Smith
Computer Sciences Corporation
NASA Advanced Supercomputing Division
NASA Ames Research Center
Moffett Field, CA 94035

Leonid Oliker
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

Rupak Biswas
NASA Advanced Supercomputing Division
NASA Ames Research Center
Moffett Field, CA 94035

Abstract

Computational grids have the potential for solving large-scale scientific problems using heterogeneous and geographically distributed resources. However, a number of major technical hurdles must be overcome before this potential can be realized. One problem that is critical to effective utilization of computational grids is the efficient scheduling of jobs. This work addresses this problem by describing and evaluating a grid scheduling architecture and three job migration algorithms. The architecture is scalable and does not assume control of local site resources. The job migration policies use the availability and performance of computer systems, the network bandwidth available between systems, and the volume of input and output data associated with each job. An extensive performance comparison is presented using real workloads from leading computational centers. The results, based on several key metrics, demonstrate that the performance of our distributed migration algorithms is significantly greater than that of a local scheduling framework and comparable to a non-scalable global scheduling approach.

1 Introduction

One of the primary goals of grid computing [1, 6] is to share access to geographically distributed heterogeneous resources in a transparent manner. There will be many benefits when this goal is realized, including the ability to execute applications whose computational requirements exceed local resources and the reduction of job turnaround time through workload balancing across multiple computing facilities. The development of computational grids and the associated middleware has therefore been actively pursued in recent years. However, many major technical (and political) hurdles stand in the way of realizing these benefits. Among the myriad research issues to be addressed is the problem of distributed resource management and job scheduling for computational grids. Although numerous researchers have proposed scheduling algorithms for parallel architectures [3, 4, 5, 7, 9, 11], the problem of scheduling jobs in a heterogeneous grid environment is fundamentally different. This is the focus of our work in this paper.

Our approach to this problem begins with defining a grid scheduling architecture that consists of autonomous local schedulers that schedule access to computer systems and grid schedulers, paired with local schedulers, that send jobs to local schedulers and migrate jobs between grid schedulers. It is important that grid scheduling be distributed for scalability and fault tolerance and it is important that local schedulers have control of local resources so that grid scheduling will be accepted by the owners of the computer systems. Our grid scheduling architecture is presented in Section 2.

Second, we propose algorithms for migrating jobs between grid schedulers. These algorithms try to migrate jobs when the wait times of compute servers rises above or falls below specific thresholds. These migration algorithms decide whether to send or receive jobs using the requirements of each job (number of CPUs, wallclock time, amount of input data, and amount of output data) the availability and performance of computer systems, and the expected network bandwidth available between systems. Our job migration algorithms are presented in Section 3.

Third, we evaluate our grid scheduling algorithms by simulating compute servers, networks, and schedulers and driving these simulations using workloads derived from trace data gathered from leading computational centers. We gather several key performance metrics during these simulations and use these metrics to compare the performance of our algorithms and reference local and centralized scheduling algorithms. The methodolgy we use to gather performance data is presented in Section 4. Our experiments show that one of our algorithms has slightly lower turn-around times than our others and that these times are 47% less than if no grid scheduling is performed. Further, we find that for our experiments with larger data sizes and lower network bandwidths, ignoring data transfers when making migration decisions can result in 690% higher turn-around times. The results of our simulations and an evaluation of these results are presented in Section 5. Finally, we present conclusions and future work in Section 6.

2 Grid Scheduling Architecture

We use a common grid scheduling architecture, shown in Figure 1, for the grid scheduling algorithms that we propose. The architecture is composed of distributed compute servers, local schedulers with local queues, and grid schedulers with grid queues. A local job is submitted to a *local scheduler* (LS) which places the job in it's *local queue* (LQ). The local scheduler removes jobs from the local queue and executes them on the local compute server. A grid job is submitted to a *grid scheduler* which places the job in it's *grid queue* (GQ). A grid scheduler gathers information from it's local scheduler and it's peer grid schedulers and decides whether to send jobs to the local scheduler, send jobs to other grid schedulers, or request jobs from other grid schedulers. One issue which we do not address in this work is how grid schedulers locate their peer grid schedulers. We expect that traditional peer-to-peer (P2P) peer location approaches that use centralized or distributed indexes can be used and we plan to examine this issue in future work.

There are a variety of grid scheduling architectures that we could have adopted. A centralized architecture with a single scheduler for multiple computer systems might be a good choice for a relatively small set of computer systems on a single machine room floor, but this approach won't scale and is not fault tolerant in a geographically distributed environment. A hierarchy where grid schedulers are organized into a tree and jobs flow up and down the tree [8] is an interesting approach, but we do not expect it to scale as well as a P2P approach. A variation of our architecture is one in which the local scheduler and grid scheduler are combined into a single scheduler. This is starting to occur as scheduling vendors adopt a grid approach to scheduling [12, 10], but these systems don't interoperate and are not yet widely used. Another approach to grid scheduling is where local scheduling is performed as usual but grid users use user-level grid schedulers to select which local schedulers to submit applications to [2]. This approach is very similar to our P2P approach, the difference being that user-level grid schedulers are seeking to optimize the execution of jobs for a single user while our grid schedulers are seeking to optimize the execution of all jobs. We believe this subtle difference results in the P2P grid scheduling approach having greater potential scheduling performance. In the end, we chose a P2P architecture with a grid scheduler co-located with each local scheduler. We believe that this approach [13] gives us the best potential scalability, fault tolerance, and scheduling performance without requiring that sites replace their local schedulers.
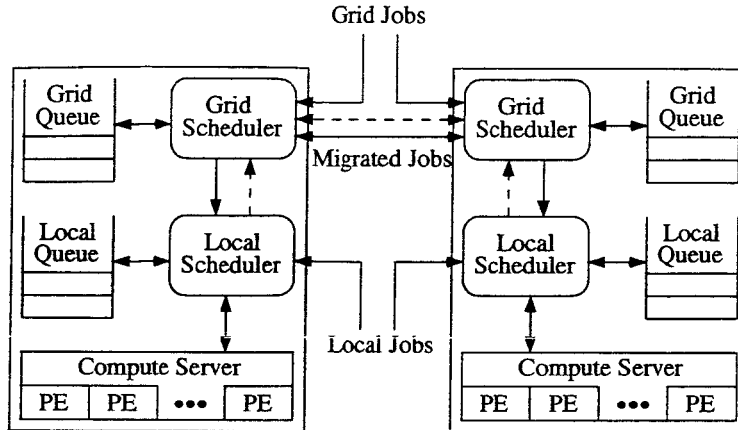
Figure 1: Our grid scheduling architecture. Solid arrows represent movement of jobs, dashed arrows represent transfer of information.

3 Grid Scheduling Algorithms

This section presents the three distributed scheduling algorithms that are the subject of this work and two reference algorithms. Our distributed scheduling algorithms are the *sender-initiated, receiver-initiated*, and *symmetrically-initiated* algorithms. These algorithms operate in a P2P manner and use different strategies for migrating jobs between grid schedulers. Our two reference algorithms that we use for comparison are a *centralized* algorithm that uses a single grid scheduler that interacts with all local schedulers and a *local* algorithm that has no grid schedulers and executes all jobs on the compute server where they were submitted.

3.1 Distributed Algorithms

Our three distributed algorithms are based around common steps:

1. A job j is submitted to a grid scheduler on compute server s and is placed in the associated grid queue.

2. The grid scheduler asks the local scheduler on s for the *approximate wait time* (AWT) of the job. The approximate wait time is the amount of time the local scheduler estimates job j, if submitted to it, will wait in the local queue before it begins executing. The AWT is computed by simulating the local scheduling algorithm using the local jobs that are either running or waiting in the local queue and the job j. If the local scheduler cannot satisfy the resource requirements of j, an AWT of infinity is returned.

3. The grid scheduler tests the approximate wait time for j against a threshold ϕ. If the AWT is less than ϕ, j is sent directly to the local scheduler for execution on s. If the AWT is at least ϕ, the job is kept in the grid queue and one of our job migration algorithms is invoked.

3.1.1 Sender-Initiated

In the sender-initiated (S-I) strategy, the grid scheduler sends the resource requirements of the job to it's peers. In this study, we only consider the CPU and run time requirements of each job; however, this can be extended to an arbitrary number of resource constraints. In response to the query, each peer grid scheduler

3

returns the *approximate turnaround time* (ATT) for the job and the *resource utilization* (RU) of the compute server associated with grid scheduler. ATT is an estimate of the amount of time it will take to complete a job and the ATT for a job j on computer server s_{exec} that is initially submitted to a grid scheduler on compute server s_{init} is derived in the following way:

$$ATT(j, s_{exec}) = max(AWT(j, s_{exec}), ADT(j_{in}, s_{init}, s_{exec})) + ERT(j, s_{exec}) + ADT(j_{out}, s_{exec}, s_{init}))$$

Before a job begins to execute, it needs to both wait in a local queue and transfer input data to the system where it will execute. $AWT(j, s_{exec})$ is the approximate wait time of job j on s_{exec} and $ADT(j_{in}, s_{init}, s_{exec})$ is the *approximate data transfer time* (ADT) of the input data of j from s_{init} to s_{exec}. We assume that these activities can be performed simultaneously so the maximum of the two constrains when the job can begin executing. The job then executes on s_{exec} with an expected run time of $ERT(j, s_{exec})$ and the output data is transferred from where it executed to the compute server where it was *initially submitted* in time $ADT(j_{out}, s_{exec}, s_{init})$. Note that the expected run time can vary from one compute server to another depending on their architectural designs and program characterizations. We simplify the calculation of ERT by assuming that run time is only related to the clock frequency of the compute server.

Resource utilization is the fraction of the computer server that is currently being utilized. We assume our compute servers have multiple CPUs that are space shared so we calculate RU as the number of CPUs assigned to jobs divided by the total number of CPUs. If certain peer grid schedulers do not respond within a specified time limit due to traffic congestion or machine failure, they are simply ignored for that request.

Based on the collected information, the grid scheduler calculates the potential *turnaround cost* (TC) of itself and each partner. To compute the optimal TC, first the minimum approximate turnaround time is found. If the minimum ATT is within a small tolerance ϵ for multiple machines, the system with the lowest resource utilization is chosen to accept the job. Thus the TC metric attempts to minimize the user's time-to-solution, while using system utilization as a tiebreaker. We found this approach to be more effective then simply relying on ATT. The job is then sent to the local scheduler (by way of it's partner grid scheduler) on the computer server with the minimal turnaround cost. Note that once a job enters a local queue, it will be scheduled and run based exclusively on the policy of the local scheduler, and can no longer migrated to another site.

3.1.2 Receiver-Initiated

The receiver-initiated (R-I) algorithm takes a more passive approach to job migration than the S-I strategy. Here, each system in the computational grid checks its own resource usage periodically at time interval σ. If the RU is below a certain threshold δ, the machine volunteers itself for receiving jobs by informing its partner set of its low utilization. Once a peer grid scheduler (say, GS_p) receives this information, it checks its grid queue for the first job waiting to be scheduled. If a job is indeed queued, its resource requirements are sent to the volunteer node. The underutilized system then responds with the job's ATT, as well as its own RU. Based on this data, GS_p computes and compares the turnaround cost between itself and the volunteer system. If the TC of the volunteer is lower than that of GS_p, the job is transferred to the LQ of that system through the GM. Otherwise, it continues to wait in the GQ until either its local AWT falls below ϕ (examined at time interval σ), or an available machine volunteers its services.

3.1.3 Symmetrically-Initiated

Unlike S-I and R-I, the symmetrically-initiated (Sy-I) algorithm works in both active and passive modes. As in the R-I strategy, each machine periodically checks its own resource usage and broadcasts a message to its partner set if it is underutilized. The difference occurs when the local approximate wait time of a job exceeds ϕ but no underutilized machine volunteers its services. In the R-I approach, the job passively

sits in the GQ while waiting for a volunteer, and periodically checks its local AWT at each σ time interval. However, the Sy-I algorithm immediately switches to active mode and sends a request to its partners using the S-I strategy. The main differences in the three job migration algorithms therefore lie in the timing of the job transfer request initiations and the destination choice for those requests.

3.2 Reference Algorithms

We use two scheduling algorithms as reference algorithms to compare our work to. The centralized algorithm has a single grid scheduler and represents a performance target for our distributed scheduling approaches. The local algorithm performs no job migration and represents the current non-grid scheduling environment.

3.2.1 Centralized

In the centralized scheduling algorithm, all jobs are submitted to a single grid scheduler which does not have an affinity to a specific local system. The GS is responsible for making global decisions and assigning each job to a specific machine. The GS tracks the status of each job and maintains up-to-date information on all available resources, allowing it to compute the turnaround cost directly, without the need for any communication. When a job arrives, the GS computes its TC for all systems, selects the one with the minimum TC, and immediately migrates the job to that system. Although communication-free resource awareness is an unrealistic assumption, it allows us to model the potential gain of a centralized architecture. However, it constitutes a single point of failure and thus suffers from a lack of reliability and fault tolerance. Additionally, this approach has severe scalability problems that may result in a performance bottleneck for large-scale grid environments.

3.2.2 Local

In the local scheduling algorithm, there are no grid schedulers. All jobs are submitted to local schedulers and execute on the compute server associated with each local scheduler. This approach represents how scheduling is currently being performed and we use it as a way to demonstrate the benifits of grid scheduling algorithms.

4 Methodology

We evaluate our grid scheduling algorithms using simulations of resources and jobs. We simuluate the submission of workloads of jobs to grid schedulers, the operation of grid and local schedulers, the transfer of job input and output data between compute servers, and the execution of jobs on compute servers. During these simulations, we gather performance information so that we can compare the various grid scheduling algorithms.

4.1 Resource Configurations

We simulate 7 different compute servers in our simulations. These systems have the identical characteristics as those located at the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory, the NASA Advanced Supercomputing Division at NASA Ames Research Center, the Lawrence Livermore National Laboratory, and the San Diego Supercomputer Center. These systems are all parallel computers some of which consist of cache-coherent Symmetric MultiProcessor (SMP) nodes interconnected by a fast proprietary network others of which are NonUniform Memory Access (NUMA)

Server	Number	CPUs	CPU Speed	Site Identifier		
Identifier	of Nodes	per Node	(MHz)	(3 site)	(6 site)	(12 site)
S_1	184	16	375	0	0	0
S_2	305	4	332	1	1	1
S_3	144	8	375	2	3	2
S_4	1024	4	600	1	0	3
S_5	64	2	250	2	2	4
S_6	512	4	400	2	5	5
S_7	128	2	250	2	5	6
S_8	144	8	375	1	2	7
S_9	1024	4	600	0	4	8
S_{10}	64	2	250	0	1	9
S_{11}	512	4	400	0	3	10
S_{12}	128	2	250	1	4	11

Table 1: Configurations of the computational servers and assignment to sites when there are 3 sites, 6 sites, or 12 sites.

shared memory systems also connected by a fast proprietary network. Both types of systems partition CPUs into nodes for management purposes and the current practice is to allocate each node to a single application so that applications do not interfere with each other. We therefore used this allocation approach in our simulation environment.

We want to use 12 compute servers to give us more options for splitting servers into sets, so we duplicated 5 of the 7 compute servers to produce a total of 12. We then split the systems into 3, 6, and 12 sets to simulate compute servers grouped into 3, 6, or 12 machine rooms at different sites. Each set has an equal number of machines and we attempted to make the computational power in each set as equal as we could. The characteristics of these systems and the sites to which they are assigned are shown in Table 1.

We also simulate the networks connecting the compute servers. We assume that all of the compute servers at a single site share a network and that each of these networks is connected to every other site network using a point-to-point network connection. When we simulate the transfer of data for a job, we simulate the use of a site network on the sending side, a point-to-point network, and a site network on the receiving side. Any of these three networks can constrain the end-to-end data transfer bandwidth. We assume that all data transfers using a network share the network bandwidth equally. We perform simulations using two different assumptions about available network bandwidth. First, we assume that 800 Mb/s is available from each site network and 40 Mb/s is available from each point-to-point network. This represents a gigabit ethernet site network and a relatively high performance Wide Area Network (WAN). Second, we assume that 80 Mb/s is available from each site network and 4 Mb/s is available from each point-to-point network. This represents a 100 megabit ethernet site network and a somewhat slower WAN.

For the experiments in this paper, we make two simplifying assumptions. First, we assume that program performance is linearly related to CPU speed. Second, even though the systems we are simulating are not all binary compatible, we assume that users have compiled their applications for each of the heterogeneous platforms. We plan to relax both of this assumptions in future work.

Workload Identifier	Start Date	End Date	Number of Jobs	Average Input Size (MB) (1,000B/CPU*sec)
W_1	03/01/2002	05/31/2002	59,623	312.7
W_2	03/01/2002	05/31/2002	22,941	300.8
W_3	03/01/2002	05/31/2002	16,295	49.6
W_4	03/01/2002	05/31/2002	8,291	237.3
W_5	03/01/2002	05/31/2002	10,543	28.9
W_6	03/01/2002	05/31/2002	7,591	236.1
W_7	03/01/2002	05/31/2002	7,251	86.5
W_8	09/01/2002	11/30/2002	27,063	42.8
W_9	09/01/2002	11/30/2002	12,666	328.3
W_{10}	09/01/2002	11/30/2002	5,236	29.3
W_{11}	09/01/2002	11/30/2002	11,804	226.5
W_{12}	09/01/2002	11/30/2002	6,911	53.7

Table 2: Characteristics of the workloads used in our performance comparison. Workload W_i is submitted to the grid scheduler on system S_i during our simulations.

4.2 Workloads

We base our workloads on trace data obtained from schedulers on 7 compute servers. Seven traces, one from each system, were recorded from March of 2002 through May of 2002. Five traces were gathered from 5 of the same 7 systems but recorded from September of 2002 through November of 2002. These 12 traces do not include information on how much input data is used by each job and how much output data is produced by each job because this data is not typically available to local schedulers. So, we added synthetic information about input and output data sizes to each job in the workloads.

When adding input and output data sizes to the jobs, we assume that the amount of this data is correlated to the amount of work (number of CPUs multiplied by amount of wallclock run time) performed by each job. We also add a random element to calculating this data so we set the amount of input data for a job j using a Gaussian distribution with a mean $\mu_j = b * cpus_j * walltime\_seconds_j$ and a standard deviation of $\sigma_j = \frac{\mu_j}{3}$ where b is the amount of bytes for each unit of work the job performs. Using anecdotal observations, our best estimate for b is 1,000 bytes for each CPU second the application executes. For comparison, we also create workloads assuming that b is 100 and 10,000. We refer to these workloads, creatively, as small data, medium data, and large data. In all cases, we assume that the output data size is 5 times as large as the input data size calculated using one of the previous methods.

The characteristics of our workloads are shown in Table 2.

4.3 Performance Metrics

We use several key metrics in our simulations to evaluate the effectiveness of our proposed grid scheduling architecture and job migration algorithms. These metrics are also used to compare performance with local and centralized job scheduling schemes.

Since individual users and system administrators often have different (and possibly conflicting) demands, no single measure can comprehensively capture overall grid performance. From the users' perspective, key measures of grid performance include the *Average Response Time* and the *Average Wait Time*. These are computed as follows (N is the total number of jobs):

7

$$\text{Average Response Time} = \frac{1}{N} \sum_{j \in Jobs} (EndTime_j - SubmitTime_j)$$

$$\text{Average Wait Time} = \frac{1}{N} \sum_{j \in Jobs} (StartTime_j - SubmitTime_j)$$

where $SubmitTime_j$, $StartTime_j$, and $EndTime_j$ are the times when job j is submitted to the queue, when it commences execution, and when it is completed. The response (or turnaround) time is probably the single most important measure for an individual submitting a job; however, the wait time is also critical to users even though it is usually beyond their control.

A system administrator (or funding agency), on the other hand, is more interested in maximizing the utilization of the available computational resources at his/her center. Thus, we present the *Weighted Utilization* metric, which measures the overall ratio between consumed and available computational resources across a grid. It is computed as:

$$\text{Weighted Utilization} = \frac{\sum_{j \in Jobs} (EndTime_j - StartTime_j) \times CPUs_j \times CPUSpeed_j}{(EndTime_{last\_job} - SubmitTime_{first\_job}) \times \sum_{m \in Servers} CPUs_m \times CPUSpeed_m} \times 100\%$$

where $(EndTime_{last\_job} - SubmitTime_{first\_job})$ is the duration of the entire simulation; $CPUs_j$ and $CPUSpeed_j$ are the number of processors used by job j and their clock speed; and $CPUs_m$ and $CPUSpeed_m$ are the number of processors in machine m and their clock speed. Individual site-specific system utilizations are also reported to understand the effects of superscheduling on local computational centers.

The metric *Fraction of Jobs Transferred* allows us to determine if there is any relationship between the number of jobs transferred and the performance of the scheduling algorithms. This metric is defined as:

$$\text{Fraction of Jobs Transferred} = \frac{\text{Number of Jobs Transferred}}{\text{Total Number of Jobs}}$$

Finally, we use the total volume of data transferred to help determine if the amount of data transferred by a scheduling algorithm is affecting it's performance. This metric is defined as:

$$\text{Data Volume} = \sum_{j \in Jobs} (InputDataSize_j + OutputDataSize_j)$$

Note that performance, measured by any metric, is highly dependent on the workload requirements. For example, we would not expect an underloaded system to derive much benefit from a superscheduler in terms of grid efficiency, as there may not be much room for improvement.

5   Results

This section presents and analyzes the simulation results of our job migration algorithms using the performance metrics described in Section 4.3.

To begin, we compare the performance of our sender-initiated, receiver-initiated, and symmetrically-initiated job migration algorithms. The performance data for these algorithms is shown in Figure 2. One of the most important metrics is the average response time because that is ultimately what users care about. We find that the S-I algorithm has the lowest average response time and this response time is 5.5% less than the response times resulting from the other two algorithms. This response time is 47% better than the response time if only local scheduling is performed and is only 0.4% worse than the response time of the
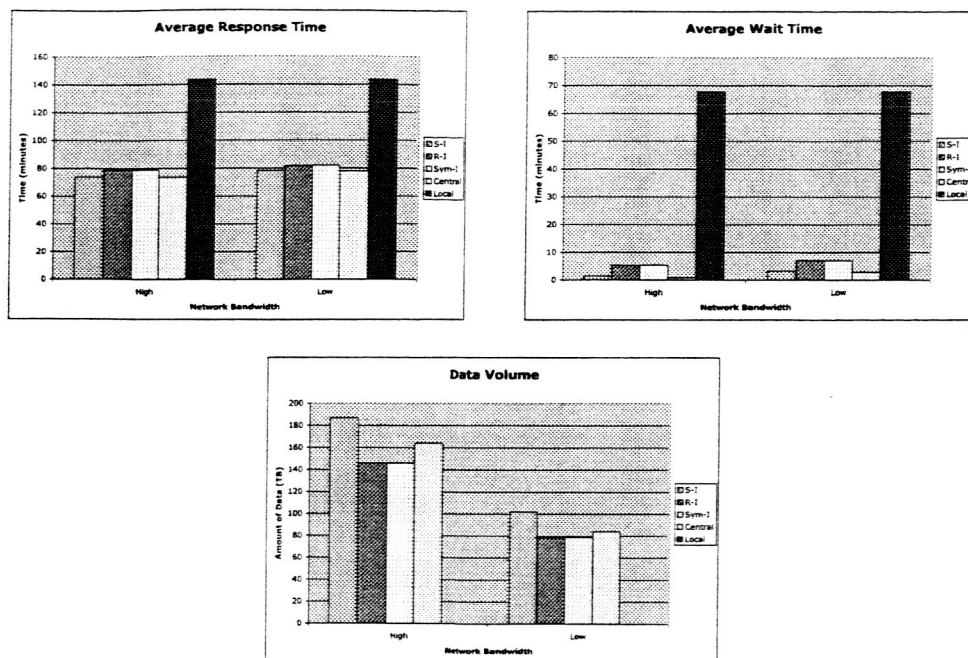
Figure 2: Comparison of the performance of our migration techniques.

centralized algorithm. We also find that the average wait times correlate with the response times: S-I has the lowest average wait time (62% less than the wait times of R-I and Sym-I) and this wait time is 34% less than the average wait time of the centralized algorithm. These differences are much more significant than the response time differences, but do not end up being significant because, on average, the wait time is only 6% of the response time.

Figure 2 also shows the average amount of data moved for each job. We find that the average data volume does not correlate with the response times: The local algorithm moves the least data and has the worst response times. The receiver-initiated and symmetrically-initiated algorithms move the next least amount of data and have the next highest response times. We do find that the centralized algorithm moves less data then the sender-initiated algorithm and also has lower response times.

A final observation is that the weighted average utilization is identical (53%) for our algorithms. This is because the jobs are submitted over time, rather than all at once, and the resource utilization obtained by even the best scheduling algorithm limited by the amount of work that is submitted to it.

The data presented in Figure 2 allows us to begin examining the effect of decreasing network bandwidth by a factor of 10. We find that this does not have a significant impact on response time or wait time but it does it does result in a 46% reduction in the amount of data transferred over the network and that 88% more jobs are executed on servers in the same site to which the they are submitted. This shows that our migration algorithms are adapting, and adapting well, to the decrease in network bandwidth. If we perform a more detailed examination and examine the performace for our small, medium, and large data workloads, we do see the effects of decreasing network bandwidth. This information for the sender-initiated algorithm is shown in Figure 3. The data shows that the large data workloads have a significant 15% increase in response time when network bandwidth is lowered while the medium data workloads only has a 2% increase and there is virtually no increase in response time for the small data workloads.

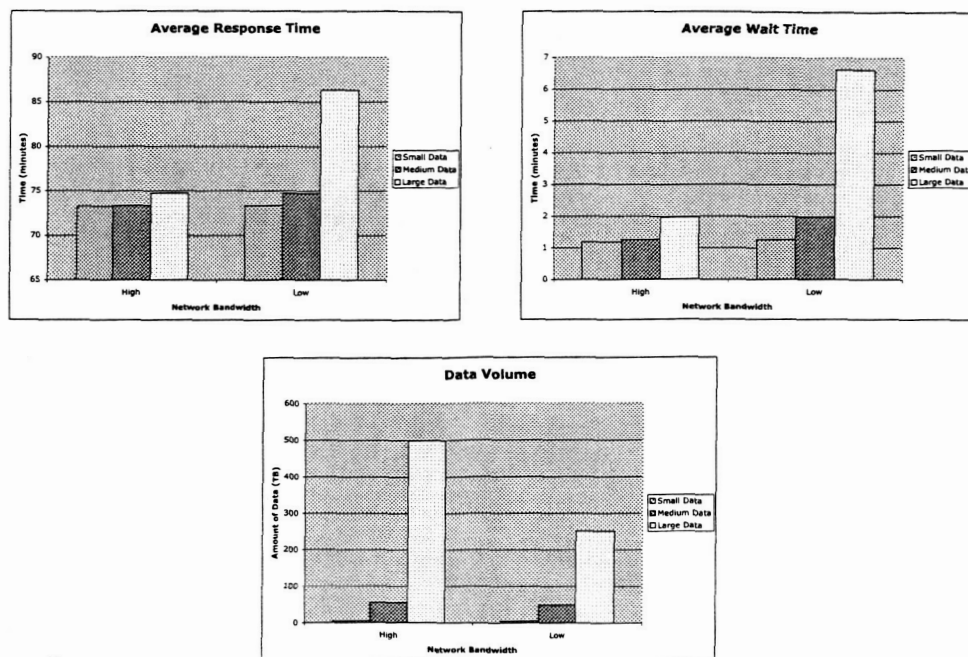We can also use Figure 3 to examine the effect of increasing the amount of data transferred per job.

Average Response Time

Average Wait Time

Data Volume

Figure 3: Performance of our sender-initiated algorithm when varing the amount of data per job.

We find that it does have a small effect when we simulate the higher network bandwidths, but the effect is more significant for lower network bandwidths. We find that the reducing data sizes from medium to small results in a reduction in response time by 2% while increasing data sizes from medium to large results in an increase in response time by 15%.

We next examine the effect of the number of sites the compute servers are grouped in to (data not shown). Contrary to our intuition, we find that the number of sites the servers are grouped in is not significant. When we examine the performance of the sender-initiated algorithm, we find that even with our large data workloads and lowest network bandwidth, having 6 sites actually reduces the average response time by 0.2% over having 3 sites and having 12 sites only increases response time over 3 sites by 0.2%.

Finally, we find that it is important to consider the size and placement of input and output data as well as the available network bandwidth when making migration decisions. We performed simulations of versions of our migration algorithms that do not consider the transfer of input and output data when making decisions. We have not completed these simulations, but for the symmetrically-initiated algorithm and workloads with large data sizes, ignoring data transfers when making decisions results in only a 4% increase in transfer times when using faster networks but results in a 690% increase in response times when using slower networks.

6 Conclusions and Future Work

One of the primary goals of grid computing is to share access to geographically distributed heterogeneous resources in a transparent manner. There will be many benefits when this goal is realized, including the ability to execute applications whose computational requirements exceed local resources and the reduction of job turnaround time through workload balancing across multiple computing facilities. We address this problem by defining a grid scheduling architecture and job migration algorithms and evaluating their performance.

Our grid scheduling architecture is a peer-to-peer architecture which, we believe, is the architecture that

will provide the best scalability and fault tolerance. Further, our architecture leaves local schedulers in place and therefore does not take over control of local resources.

We propose three job migration algorithms; the sender-initiated algorithm sends jobs from overloaded compute servers to less loaded servers, the receiver-initiated algorithm requests jobs from underloaded compute servers, and the symmetrically-initiated algorithm uses a combination of both approaches. Our experiments show that our sender-initiated algorithm has over 5% lower turn-around times than our others and that these times are 47% less than if no grid scheduling is performed. Further, we find that for our experiments with larger data sizes and lower network bandwidths, ignoring data transfers when making migration decisions can result in 690% higher turn-around times.

There are several areas of future work that we plan to explore. We wish to study how our grid scheduling scales to a large number of grid schedulers, including addressing problems such as how grid schedulers find peers. We plan to relax assumptions such as performance being related only to CPU speed and that every application can execute on every system. We also wish to compare our grid scheduling approach to others such as hierarchical and when grid and local schedulers are combined.

References

[1] Global Grid Forum. http://www.gridforum.org.

[2] Fran Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Supercomputing '96*, 1996.

[3] D.G. Feitelson. Packing schemes for gang scheduling. In *2nd Workshop on Job Scheduling Strategies for Parallel Processing*, volume LNCS 1162, pages 89–100, 1996.

[4] D.G. Feitelson, L. Rudolph, U. Schwiegelshohn, K.C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *3rd Workshop on Job Scheduling Strategies for Parallel Processing*, volume LNCS 1291, pages 1–34, 1997.

[5] D.G. Feitelson and A.M. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *12th International Parallel Processing Symposium*, pages 542–546, 1998.

[6] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, 1999.

[7] H. Franke, J. Jann, J.E. Moreira, P. Pattnaik, and M.A. Jette. A evaluation of parallel job scheduling for ASCI Blue-Pacific. In *SC99 Conference*, CD-ROM, 1999.

[8] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Evaluation of job-scheduling strategies for grid computing. In *1st International Workshop on Grid Computing*, volume LNCS 1971, pages 191–202, 2000.

[9] J. Krallmann, U. Schwiegelshohn, and R. Yahyapour. On the design and evaluation of job scheduling algorithms. In *5th Workshop on Job Scheduling Strategies for Parallel Processing*, volume LNCS 1659, pages 17–42, 1999.

[10] The Load Sharing Facility. http://www.platform.com/products/LSFfamily/.

[11] R.D. Nelson, D.F. Towsley, and A.N. Tantawi. Performance analysis of parallel processing systems. *IEEE Transactions on Software Engineering*, 14(4):532–540, 1988.

[12] The Portable Batch System. http://www.pbspro.com.

[13] Hongzhang Shan, Leonid Oliker, and Rupak Biswas. Job Superscheduler Architecture and Performance in Computational Grid Environments. In *SC2003 Conference*, 2003.